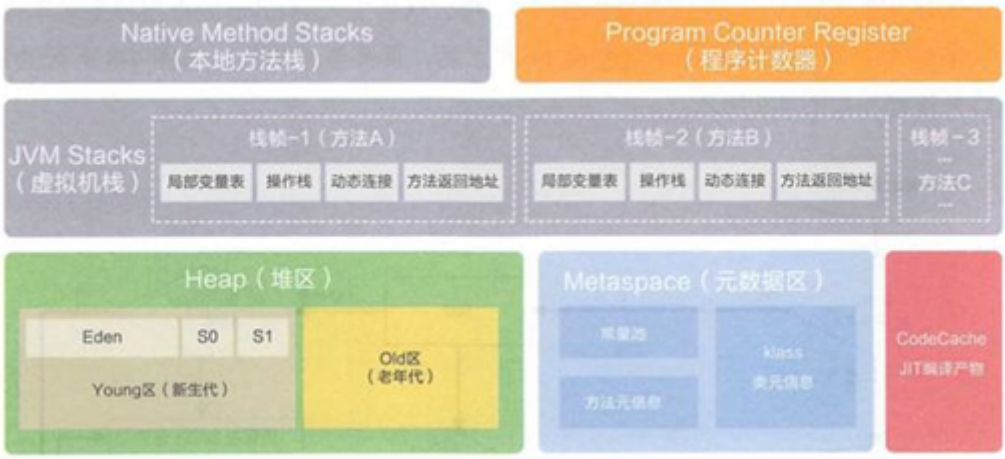
基本概念

1. 作用：屏蔽各种硬件和操作系统内存的访问差异，以实现让Java程序在各种平台下都能达到一致的内存访问效果，定义了Java虚拟机与计算机内存是如何协同工作的，是一种主存与工作内存的抽象概念，并且底层对应CPU的寄存器、缓存、硬件内存以及CPU指令优化等
2. 简称：Java Memory Model简称JMM
3. 目的：定义程序中各个变量的访问规则，以及在必须时如何同步地访问共享变量，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节

JVM内存模型如下图



1. **Head（堆）：**
   - 运行时的数据区，堆是由垃圾回收机制来负责的。
   - **运行时动态分配内存大小，并且由GC动态完成内存的回收**
   - **存取速度相对要慢**
2. **Stack（栈）：**
   - 存取速度比堆要快，仅次于计算机里的寄存器
   - 数据可共享
   - 栈中的数据的大小以及生存期必须是确定的，缺乏一些灵活性
   - **栈中主要用来存储一些基本数据类型和引用数据类型**

Java内存模型要求调用栈和本地变量存放在线程栈（Thread Stack）上，而对象则存放在堆上。一个本地变量也可能是指向一个对象的引用，这种情况下这个保存对象引用的本地变量是存放在线程栈上的，但是对象本身则是存放在堆上的。一个对象可能包含方法，而这些方法可能包含着本地变量，这些本地变量仍然是存放在线程栈上的。即使这些方法所属的对象是存放在堆上的。一个对象的成员变量，可能会随着所属对象而存放在堆上，不管这个成员变量是原始类型还是引用类型。**静态成员变量则是随着类的定义一起存放在堆上**。存放在堆上的对象，可以被持有这个对象的引用的线程访问。当一个线程可以访问某个对象时，它也可以访问该对象的成员变量。如果两个线程同时调用同一个对象上的同一个方法，那么它们都将会访问这个方法中的成员变量，但是每一个线程都拥有这个成员变量的私有拷贝。

JMM 体现在以下几个方面：

1. **原子性** - 保证指令不会受到线程上下文切换的影响
2. **可见性** - 保证指令不会受 cpu 缓存的影响
3. **有序性** - 保证指令不会受 cpu 指令并行优化的影响

java中多线程通信基于共享内存的方式，并且JMM保证了共享内存的可见性和有序性的问题，锁解决了原子性的问题。

Java内存模型和硬件内存架构之间的桥接

1. 硬件内存架构中不区分线程栈和堆，所有的数据都存储在主内存中或CPU缓存中和CPU内部的寄存器中
2. JMM中区分主内存（线程共享）和线程工作内存（线程私有，JMM中的抽象概念）
3. **Java内存模型中的线程的工作内存（working memory）是cpu的寄存器和高速缓存的抽象描述。主内存则可理解为物理主存的抽象**
4. JVM内存模型只是一种对物理内存的划分而已，它只局限在物理内存，而且只局限在JVM进程中的的物理内存。举个例子，A和线程B要通信，必须经历两个步骤:
    - 首先线程A要把本地内存A中更新过的共享变量刷新到主内存里
    - 然后线程B再到主内存中去读取线程A更新的共享变量，这样就完成了两个线程之间的通信了

Java内存模型定义同步的八种操作

JLS定义了线程对主存的操作指令：`lock, unlock, read, load, use, assign, store, write`。这些行为是不可分解的原子操作，在使用上相互依赖，

1. read-load从主内存复制变量到当前工作内存，
2. use-assign执行代码改变共享变量值，
3. store-write用工作内存数据刷新主存相关内容。
4. read（读取）：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的load动作使用
5. load（载入）：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中。
6. use（使用）：作用于工作内存的变量，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。
7. assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
8. store（存储）：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的write的操作。
9. write（写入）：作用于主内存的变量，它把store操作从工作内存中一个变量的值传送到主内存的变量中。

多线程的三个特性

原子性（Atomicity）

1. 一个原子操作在CPU中不可以中断或暂停后在调度，全执行或全不执行，但是x++不是原子操作
    - a.将变量x 值取出放在寄存器中
    - b.将将寄存器中的值+1
    - c.将寄存器中的值赋值给x。
2. JMM中的原子性操作：`read、load、use、assign、store和write`，大致可以认为基础数据类型的访问和读写是具备原子性的
3. 更大范围的操作保证原子性：在JMM中针对synchronized关键字增加了字节码指令monitorenter和monitorexit
4. 业务代码控制：锁（lock和unlock）

可见性(Visibility)

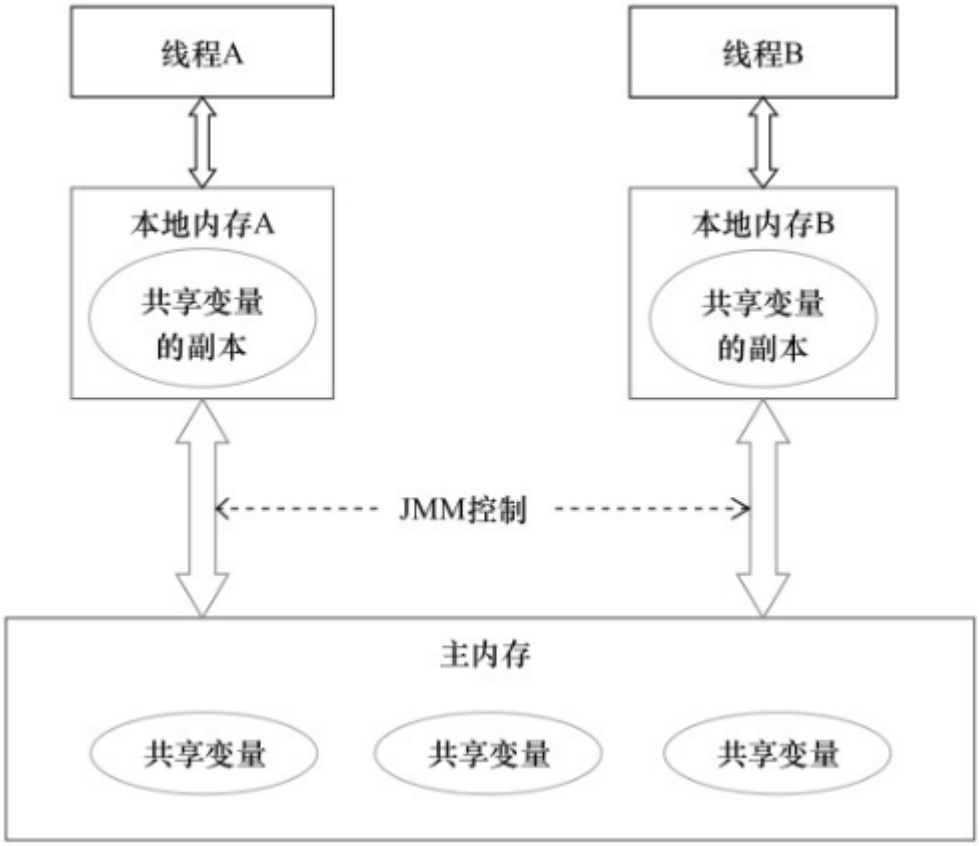
1. 使用主内存和工作内存保证可见性
2. 关键字：`volatile`，具体说明，参考后续章节

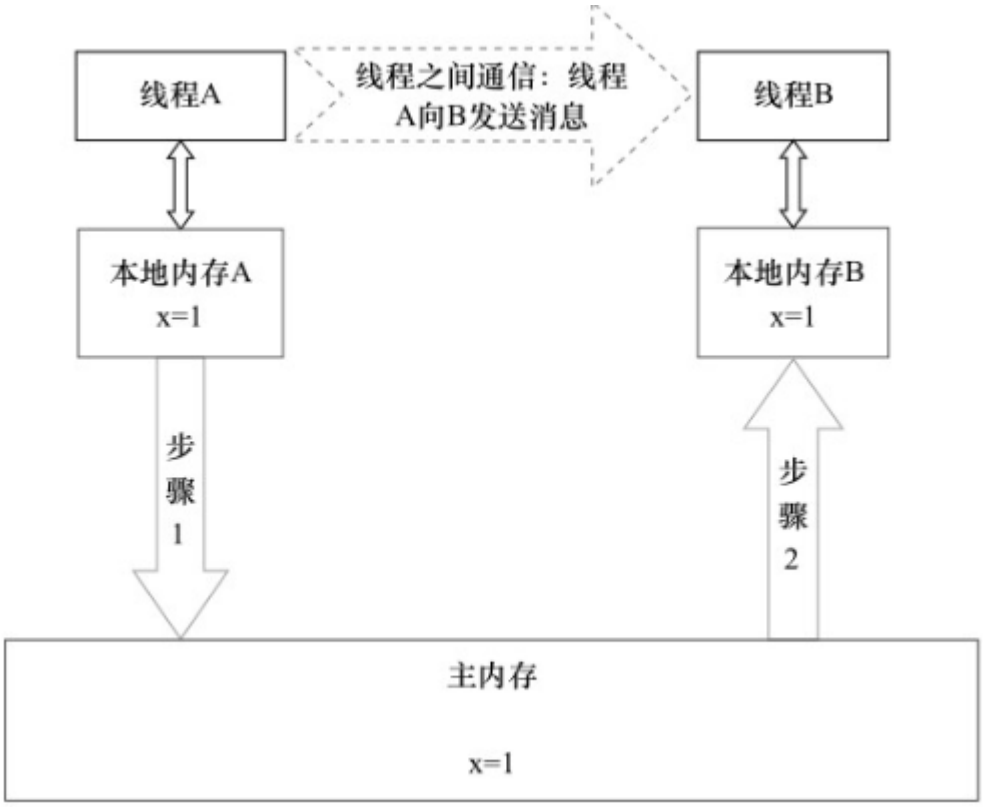
有序性(Ordering)

Java内存模型中的程序天然有序性可以总结为一句话：如果在本线程内观察，所有操作都是有序的；如果在一个线程中观察另一个线程，所有操作都是无序的。前半句是指“线程内表现为串行语义”，后半句是指“指令重排序”现象和“工作内存主主内存同步延迟”现象。

JMM抽象结构

1. 共享数据：实例域、静态域、数组存储在堆中，线程间共享
2. 非共享数据：局部变量、方法参数、异常处理器参数





happens-before

happens-before仅仅要求前一个操作（执行的结果）对后一个操作可见；规定了对共享变量的写操作对其它线程的读操作可见，它是可见性与有序性的一套规则总结

1. ❑ 程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作。
2. ❑ 监视器锁规则：线程解锁 m 之前对变量的写，对于接下来对 m 加锁的其它线程对该变量的读可见
3. ❑ volatile变量规则：线程对 volatile 变量的写，对接下来其它线程对该变量的读可见
4. ❑ 线程 start 前对变量的写，对该线程开始后对该变量的读可见
5. ❑ 线程结束前对变量的写，对其它线程得知它结束后的读可见
6. ❑ 线程 t1 打断 t2（interrupt）前对变量的写，对于其他线程得知 t2 被打断后对变量的读可见
7. ❑ 对变量默认值（0，false，null）的写，对其它线程对该变量的读可见
8. ❑ 传递性：如果A happens-before B，且B happens-before C，那么A happens-before C。

重排序

顺序一致性

volatile

volatile的作用

1. **同步**：一种弱的同步机制，用来确保将变量的更新操作通知到其他线程，Volatile修饰变量的单个读/写，可以看做使用同一个锁完成的同步操作。
2. **可见性**
   - **读**： 执行**read->load->use**，每次使用时都需要从主内存中获取最新的数据
   - **写**： 执行**assign->store->write**，每次更新完后都会回写到主内存
3. **禁止指令重排序**

4. **场景**：一个变量被多个线程共享，线程直接给这个变量赋值。
5. **保证线程安全的前置条件**
    - 单个读/写
    - 不同的volatile变量之间，不能互相依赖
6. **原子性**：只能对原子性（单个读/写）操作具备线程安全性
7. **原理**：普通变量的获取首先把主内存的数据获取到CPU缓存中，然后通过缓存使用；被volatile修饰的变量JVM保证了每次读变量都从内存中读，跳过 CPU cache 这一步，直接通过主内存获取数据

代码分析

```
public class LazySingleton {
    private static volatile LazySingleton instance = null;
    public static LazySingleton getInstance() {
        if (instance == null)
            instance = new LazySingleton();
        return instance;
    }
    public static void main(String[] args) {
        LazySingleton.getInstance();
    }
}
```

将代码转换为汇编指令，在汇编指令的中我们可以找到如下信息

```
0x0000000002931351: lock add dword ptr [rsp],0h ;*putstatic instance--将双字节的栈指针寄存器+0
; - org.xrq.test.design.singleton.LazySingleton::getInstance@12 (line 13) --赋值
```

在使用volatile关键字修饰之后JMM会增加一个**lock前缀指令的内存屏障**，完成以下功能：

1. 禁止指令重排序
2. 它会强制将对缓存的修改操作立即写入主存；
3. 如果是写操作，它会导致其他CPU中对应的缓存行无效。

> 这里需注意虽然volatile关键字保证了变量对于线程的可见性，但并不保证线程安全。

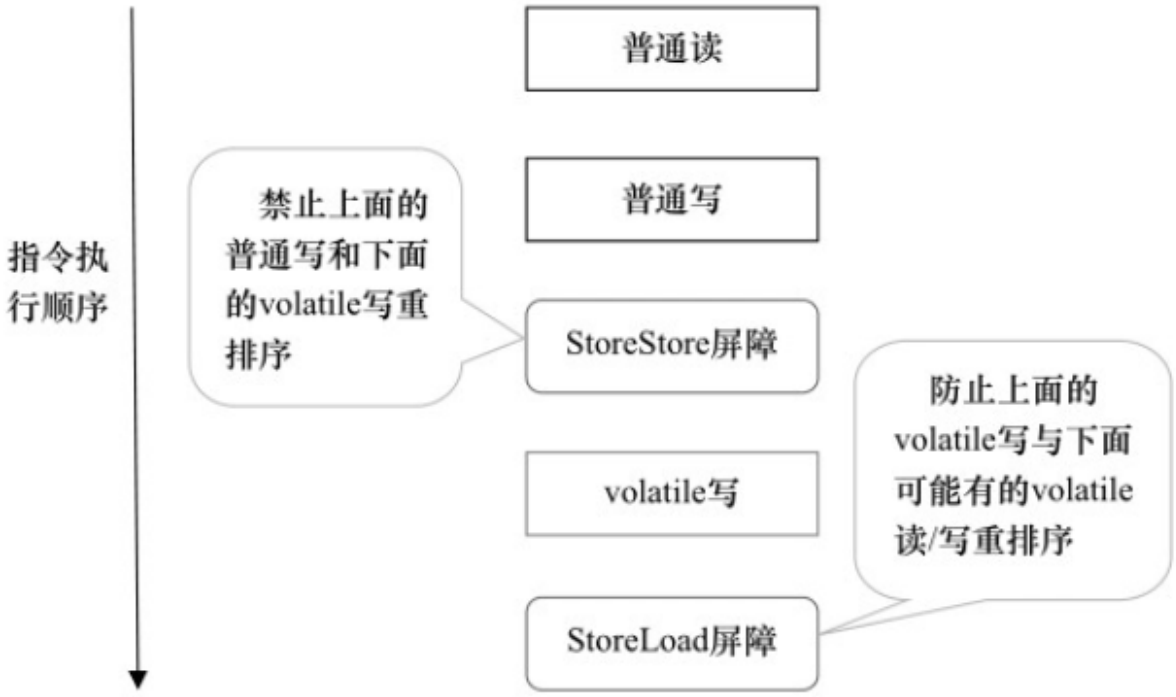
volatile内存语义的实现

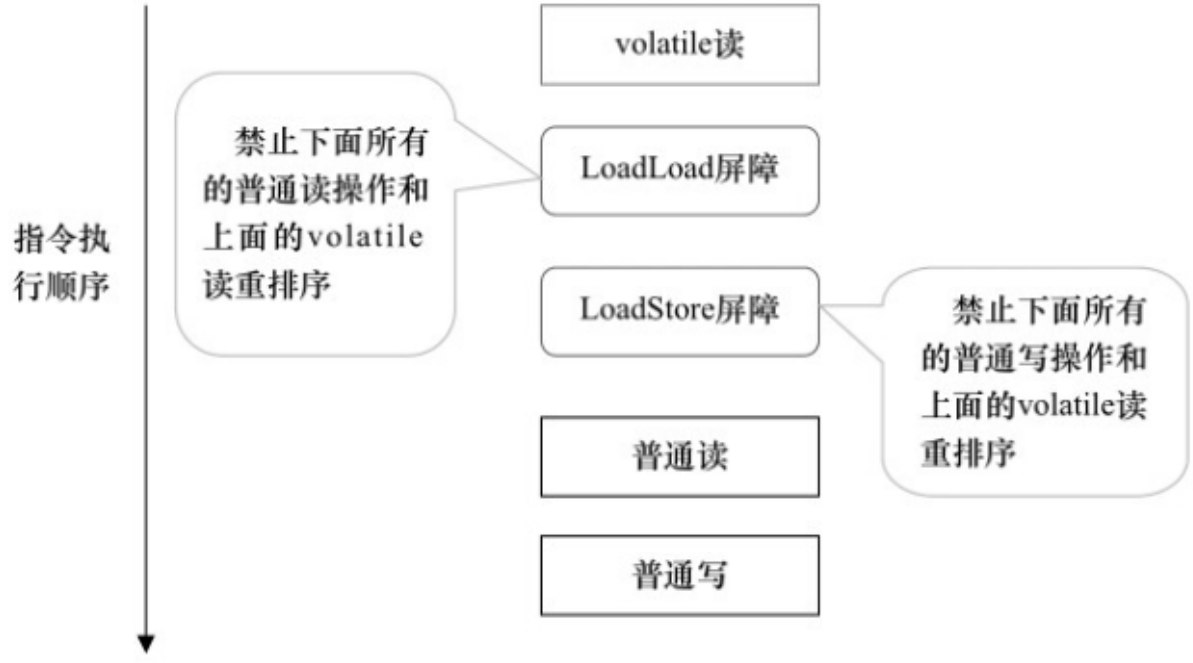
是否能重排序	第二个操作		
第一个操作	普通读 / 写	volatile 读	volatile 写
普通读 / 写			NO
volatile 读	NO	NO	NO
volatile 写		NO	NO

内存屏障的使用

基于保守策略的JMM内存屏障插入策略。

1. ❑ 在每个volatile写操作的前面插入一个StoreStore屏障。
2. ❑ 在每个volatile写操作的后面插入一个StoreLoad屏障。
3. ❑ 在每个volatile读操作的前面插入一个LoadLoad屏障。
4. ❑ 在每个volatile读操作的后面插入一个LoadStore屏障





死循环代码

```java
package com.sunld.thread.jmm;

import java.util.concurrent.atomic.AtomicInteger;

import static java.lang.Thread.sleep;

/**
 * @author : sunliaodong
 * @version : V1.0.0
 * @description: 不适用Volatile导致的死循环
 * 不同的CPU可能无法达到演示效果
 * @date : 2020/5/31 16:05
 */
public class VolatileDeadCircleTest {
    private volatile boolean run = true;

    public static void main(String[] args) throws InterruptedException {
        VolatileDeadCircleTest t = new VolatileDeadCircleTest();
        AtomicInteger count = new AtomicInteger();
        new Thread(() -> {
            while(t.isRun()){
                try {
                    sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("子线程一致执行:" + (count.getAndIncrement()));
            }
        }).start();
        sleep(10 * 1000L);
        // 计划停止子线程，但是和预期可能不一致，最终会停止
        t.setRun(false);
    }

    public boolean isRun() {
        return run;
    }

    public void setRun(boolean run) {
        this.run = run;
    }
}
```

锁的内存语义synchronized

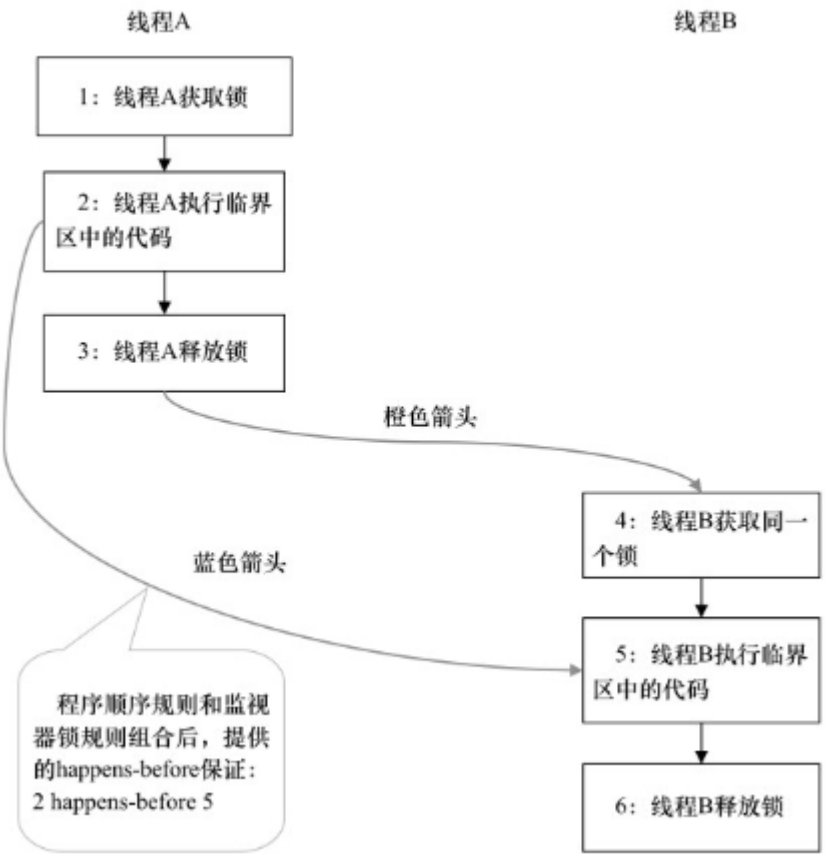
1. 锁可以让临界区互斥执行
2. 重要的同步手段
3. 释放锁的线程向获取同一个锁的线程发送消息（唤醒）

锁的释放、获取

```
package com.sunld.thread.jmm;

/**
 * @author : sunliaodong
 * @version : V1.0.0
 * @description: TODO
 * @date : 2020/6/3 17:34
 */
public class MonitorExample {
    int a = 0;

    public synchronized void writer(){ //1
        a++;                           //2
    }
                                       //3
    public synchronized void reader(){ //4
        System.out.println(a);         //5
    }
                                       //6
}
```



1. 线程之间使用同一个锁存在先后顺序
2. 某个方法使用完成之后释放对应的锁，其他线程可竞争获取该锁
3. synchronized使用的是监视器锁
4. 锁释放与volatile写有相同的内存语义；锁获取与volatile读有相同的内存语义（释放锁之后会把共享数据刷入主存并且使其他线程的本地内存中的变量无效）

锁内存语义实现

final的内存语义

对final域的读写更像是普通变量访问。

final域的重排序规则

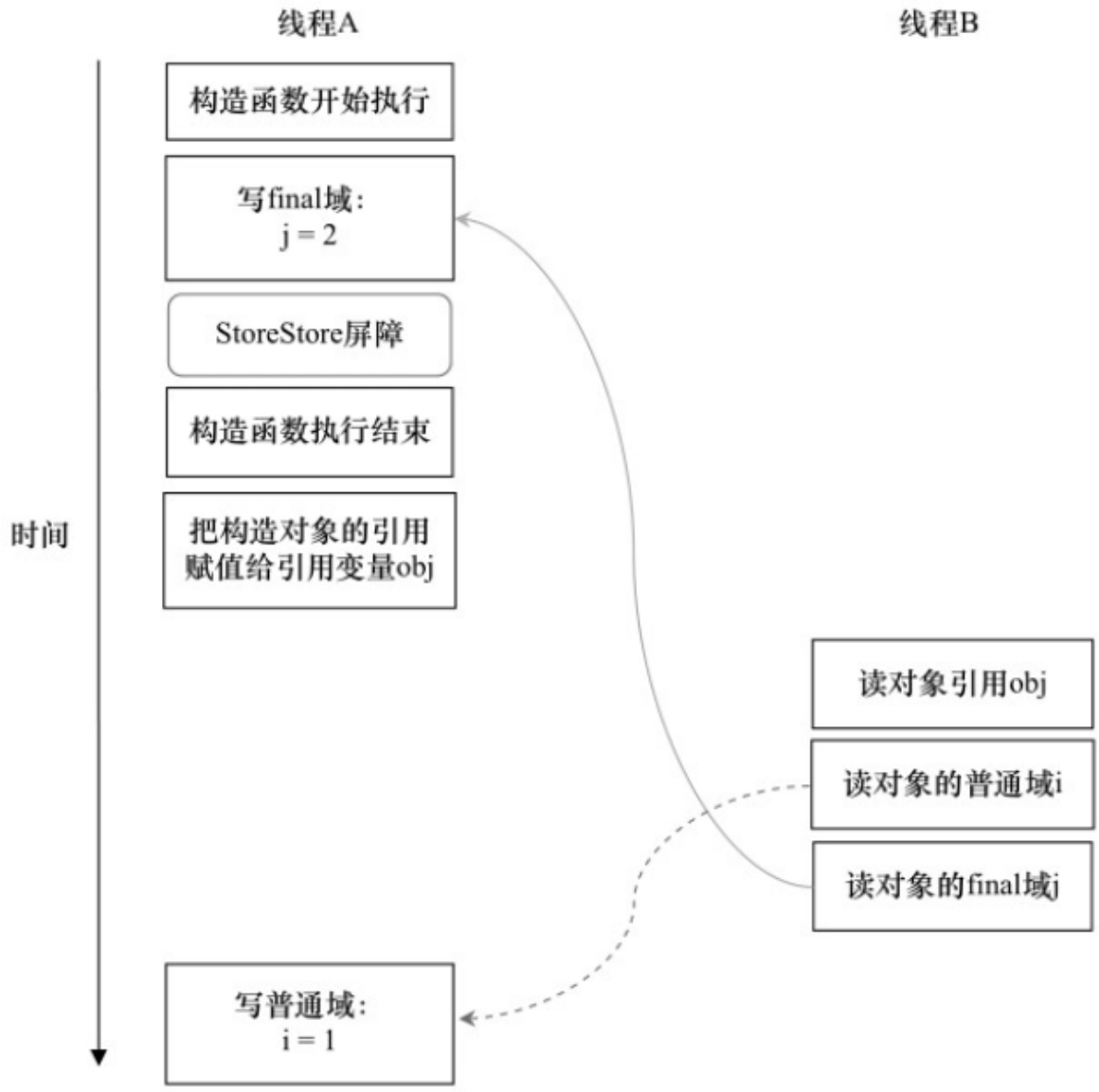
1. 在构造函数内对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。
2. 初次读一个包含final域的对象的引用，与随后初次读这个final域，这两个操作之间不能重排序。

```
package com.sunld.thread.jmm;

/**
 * @author : sunliaodong
 * @version : V1.0.0
 * @description: TODO
 * @date : 2020/5/31 21:04
 */
public class FinalExample {
    int i;                            // 普通变量
    final int j;                      // final变量
    static FinalExample obj;
    public FinalExample(){            // 构造函数
        i = 1;                        // 普通变量赋值
        j = 2;                        // final变量赋值
    }
    public static void writer(){    // 写线程A执行
        obj = new FinalExample();
    }
    public static void reader(){    // 读线程B执行
        FinalExample object = obj; // 读对象引用
        int a = object.i;           // 读普通变量
        int b = object.j;           // 读final变量
    }
}
```

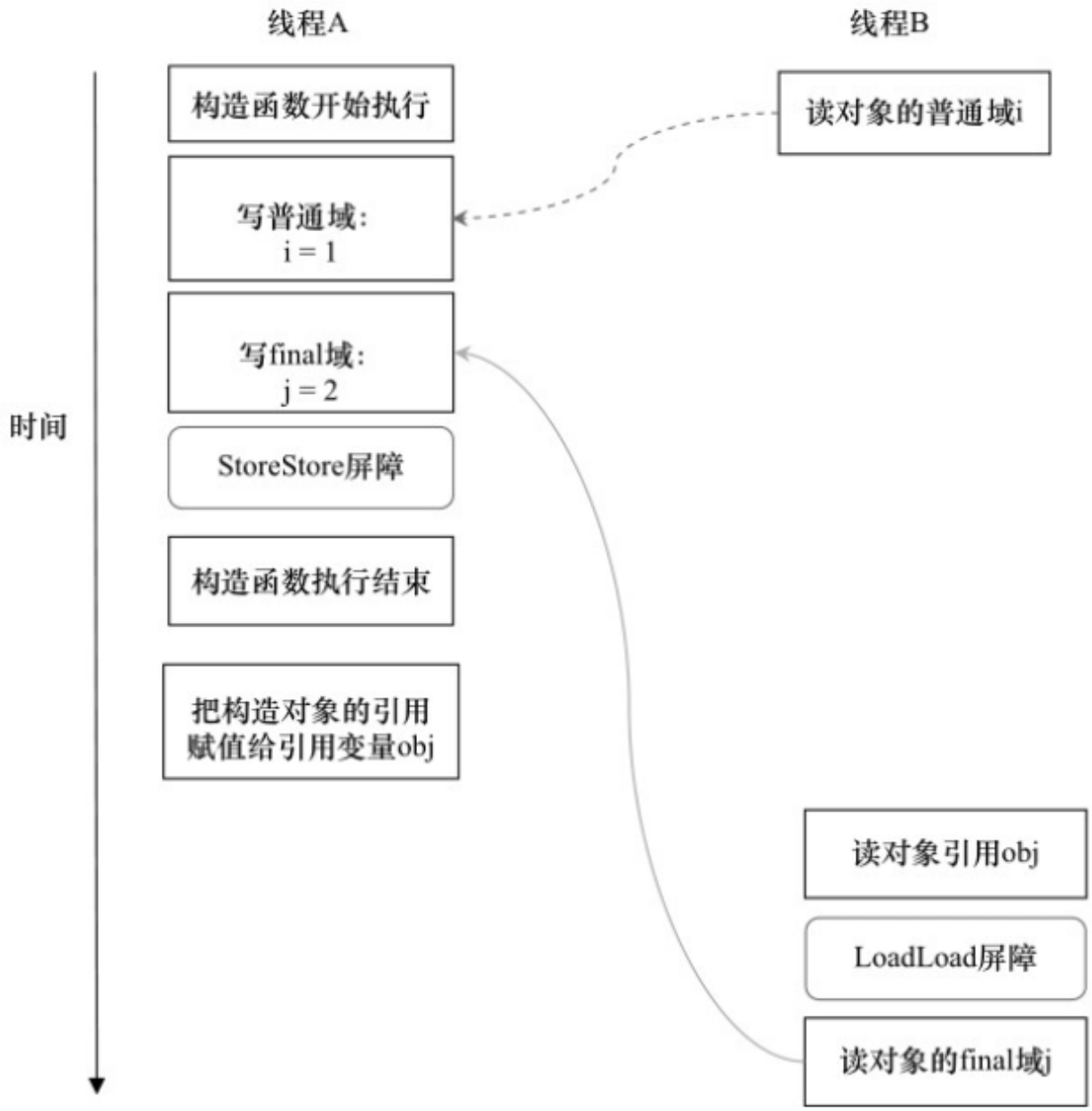
写final域的重排序规则

1. JMM禁止编译器把final域的写重排序到构造函数之外
2. 编译器会在final域的写之后，构造函数return之前，插入一个StoreStore屏障。这个屏障禁止处理器把final域的写重排序到构造函数之外。
3. 在对象引用为任意线程可见之前，对象的final域已经被正确初始化过了，而普通域不具有这个保障
4. 写final域的重排序规则会要求编译器在final域的写之后，构造函数return之前插入一个StoreStore障屏。读final域的重排序规则要求编译器在读final域的操作前面插入一个LoadLoad屏障。



读final域的重排序规则

编译器会在读final域操作的前面插入一个LoadLoad屏障，防止初次读对象引用与初次读该对象包含的final域出现重排序。



final域为引用类型

在构造函数内对一个final引用的对象的成员域的写入，与随后在构造函数外把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。

final引用不能从构造函数内“溢出”

在构造函数内部，不能让这个被构造对象的引用为其他线程所见，也就是对象引用不能在构造函数中“逸出”。

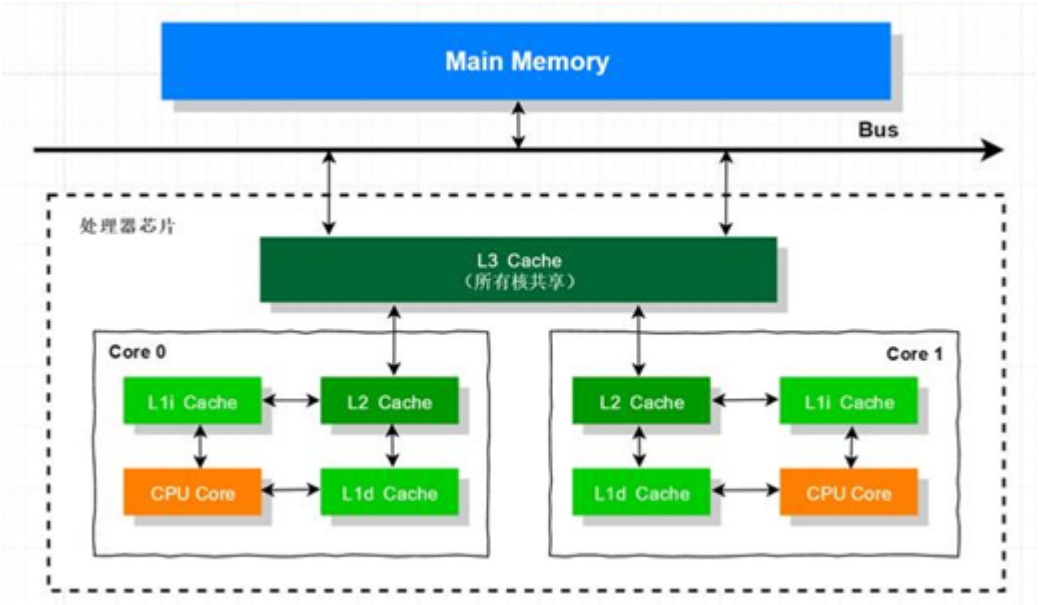
CPU高速缓存详解

背景

JVM模型中的主要概念：

1. **线程共享（线程不安全）：堆内存、全局变量、主内存**
2. **线程私有（线程安全）：本地内存，栈内存，局部变量**

线程共享数据的使用是由**硬件架构**决定，JVM针对硬件做了更高层级的封装和抽象，并且提供并发场景下的安全保证和API。java语言层面使用**Synchronize， Volatile**等关键字实现。



硬件内存模型

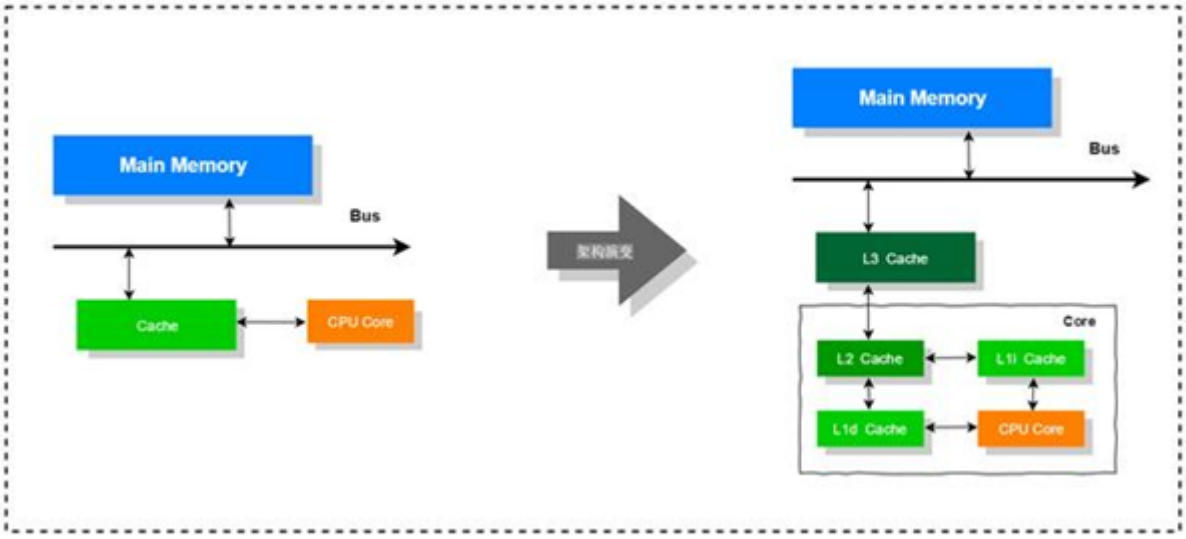
1. 引入CPU缓存的目的：**解决CPU运算速度与内存读写速度不匹配**，加快读取速度，提高CPU的利用率
2. 位置：位于CPU与内存之间的临时存储器，对比主存内存空间小，但是交换速度快

缓存分类

1. **一级缓存（L1 Cache）：一级数据缓存（Data Cache， D-Cache， L1d）和一级指令缓存(Instruction Cache， I-Cache， L1i)，分别用于存放数据和指令。**
2. **二级缓存（L2 Cache）：分内部和外部两种芯片。内部的芯片二级缓存运行速度与主频相同，而外部的二级缓存则只有主频的一半。L2高速缓存容量也会影响CPU的性能，原则是越大越好，现在家庭用CPU容量最大的是4MB，而服务器和工作站上用CPU的L2高速缓存普遍大于4MB，有的高达8MB或者19MB。**
3. **三级缓存（L3 Cache）：在拥有三级缓存的CPU中，只有约5%的数据需要从内存中调用，这进一步提高了CPU的效率。具有较大L3缓存的处理器提供更有效的文件系统缓存行为及较短消息和处理器队列长度。**

工作原理

1. 读取数据的流程：**CPU读取数据的顺序是先Cache后内存（结构：CPU -> cache -> memory**）
2. Cache命中率可达**90%左右**
3. CPU Cache的意义：提高吞吐量和CPU利用率
4. 局部性原理
    - 时间局部性：如果某个数据被访问，那么在不久的将来它很可能再次被访问
    - 空间局部性：如果某个数据被访问，那么与它相邻的数据很快也可能被访问



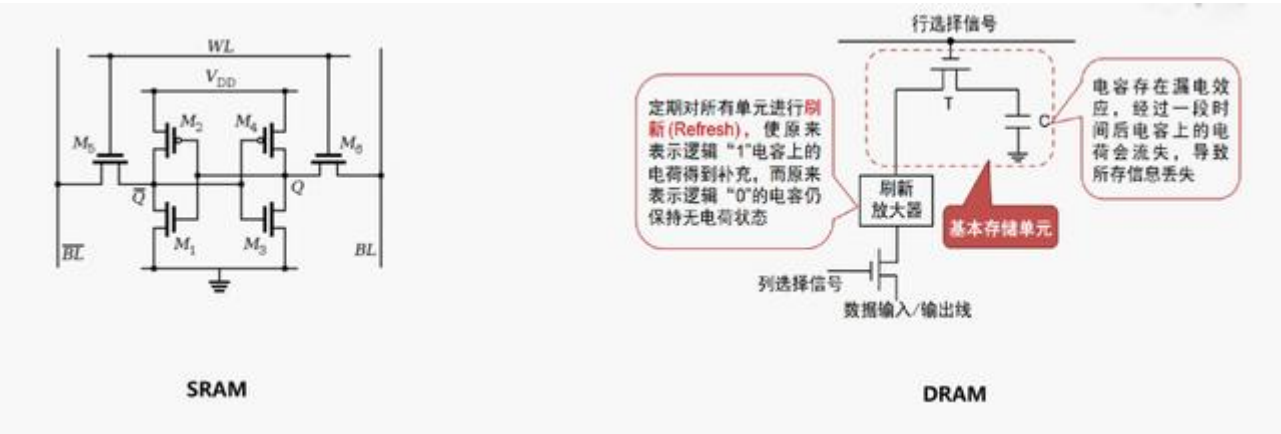
缓存SRAM与内存DRAM的区别

内存的DRAM其实是SDRAM（同步动态随机储存器），是DRAM（Dynamic RAM，动态）的一种。DRAM只含一个晶体管和一个电容器，集成度非常高，可以轻松做出**大容量（内存）**，但是因为靠电容器来储存信息，所以需要不断刷新补充电容器的电荷，否则内部的数据即会消失。

充电放电之间的时间差导致了DRAM比SRAM的反应要缓慢得多。**高速缓存基本上都是采用SRAM存储器,** SRAM是英文Static RAM的缩写，它是一种具有静态存取功能的存储器，不需要刷新电路即能保存它内部存储的数据。

SRAM相比DRAM的复杂度就高了不止一点点，所以导致SRAM的集成度很低，也是前期CPU缓存不能集成进CPU内部也有这个原因。因此相同容量的DRAM内存可以设计为较小的体积，但是SRAM却需要很大的体积，这也是不能将缓存容量做得太大的重要原因。

> 它的特点归纳如下：优点是节能、速度快、不需要刷新时间所以凸显其数据传输速度很快，缺点是集成度低、相同的容量体积较大、而且价格较高，只能少量用于关键性系统以提高效率。SRAM和DRAM的电路图大致如下：



在一套完整的计算机体系中，一般会包含如下存储介质，它们分别是**寄存器，高速缓存，内存，硬盘**。它们的容量由小到大，访问速度由高到底，成本由高到低，体积则是由小到大。因此，寄存器，高速缓存等容易集成在CPU或是主板上，但是由于其成本高，所以不会大规模使用，而内存，硬盘成本较低，但体积大，不方便集成，所以可以作为外设，安装在主板上。这些存储介质的大小和访问速度大致如下图：

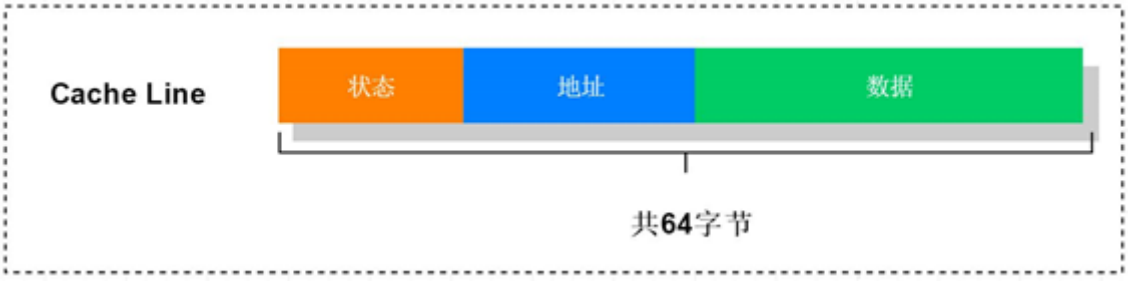
缓存一致性（MESI）

在多核CPU中，内存中的数据会在多个核心中存在数据副本，某一个核心发生修改操作，就产生了数据不一致的问题。而一致性协议正是用于保证多个CPU cache之间缓存共享数据的一致。试想下面一个问题：

1. Core 0读取了一个字节，根据局部性原理，它相邻的字节同样被被读入Core 0的缓存
2. Core 1做了上面同样的工作，这样Core 0与Core 1的缓存拥有同样的数据
3. Core 0修改了那个字节，被修改后，那个字节被写回Core 0的缓存，但是该信息并没有写回主存
4. Core 1访问该字节，由于Core 0并未将数据写回主存，数据不同步

缓存行

高速缓存其实就是一组称之为**缓存行(cache line)的固定大小的数据块**，其大小是以突发读或者突发写周期的大小为基础的。它是CPU缓存中可分配的**最小存储单元**，大小32字节、64字节、128字节不等，这与CPU架构有关，通常来说是64字节。当从CPU从内存中取数据到cache中时，会一次取一个cacheline大小的内存大小到cache中，然后存进相应的cacheline中。总结来说就是一句话：CPU不是按字节访问内存，而是以64字节为单位的块(chunk)拿取，称为一个缓存行(cache line)。



> 工作方式：当CPU从cache中读取数据的时候，会比较地址是否相同，如果相同则检查cache line的状态，再决定该数据是否有效，无效则从主存中获取数据，或者根据一致性协议发生一次cache-to—chache的数据推送；
> 工作效率：当CPU能够从cache中拿到有效数据的时候，消耗几个机器周期，如果发生cache miss（缓存未命中），则会消耗几十上百个机器周期；

MESI缓存一致性

CPU制造商制定了一个规则：**当一个CPU修改缓存中的字节时，服务器中其他CPU会被通知，它们的缓存将视为无效。**于是，在上面的情况下，Core 1发现自己的缓存中数据已无效，Core 0将立即把自己的数据写回主

存，然后Core 1重新读取该数据，这个就是多级缓存-**缓存一致性（MESI）。这协议用于保证多个CPU cache之间缓存共享数据的一致性。**它定义了CacheLine的四种数据状态，而CPU对cache的四种操作可能会产生不一致的状态。因此缓存控制器监听到本地操作与远程操作的时候需要对地址一致的CacheLine状态做出一定的修改，从而保证数据在多个cache之间流转的一致性。

嗅探（snooping）协议，它的基本思想是：所有内存的传输都发生在一条共享的总线上，而所有的处理器都能看到这条总线：缓存本身是独立的，但是内存是共享资源，所有的内存访问都要经过仲裁（**同一个指令周期中，只有一个CPU缓存可以对内存读写**）。CPU缓存控制器不仅仅在内存传输的时候才与总线打交道，而是不停在嗅探总线上发生的数据交换，跟踪其他缓存在做什么。所以当一个缓存代表它所属的处理器去读写内存时，其它处理器都会得到通知，它们以此来使自己的缓存保持同步。**只要某个处理器一写内存，其它处理器马上知道这块内存在它们的缓存段中已失效。**ESI协议是当前最主流的缓存一致性协议，在MESI协议中，每个缓存行有4个状态，可用2个bit表示，它们分别是：

> **M（modify）**：该Cache line有效，数据被修改了，和内存中的数据不一致，数据只存在于本Cache中。总线监听：缓存行必须时刻监听所有试图读该缓存行相对就主存的操作，这种操作必须在缓存将该缓存行写回主存并将状态变成S（共享）状态之前被延迟执行。
> **E（exclusive）**：该Cache line有效，数据和内存中的数据一致，数据只存在于本Cache中。总线监听：缓存行必须监听其它缓存读主存中该缓存行的操作，一旦有这种操作，该缓存行需要变成S（共享）状态。
> **S（shared）**：该Cache line有效，数据和内存中的数据一致，数据存在于很多Cache中。总线监听：缓存行也必须监听其它缓存使该缓存行无效或者独享该缓存行的请求，并将该缓存行变成无效（Invalid）。
> **I（invalid）**：该Cache line无效。总线监听：无

只有当缓存行处于E或者M状态时，处理器是**独占**这个缓存行的。当处理器想写某个缓存行时，如果它没有独占权，它必须先发送一条"我要独占权"的请求给总线，这会通知其它处理器把它们拥有的同一缓存段的拷贝失效（如果有）。只有在获得独占权后，处理器才能开始修改数据—-并且此时这个处理器知道，这个缓存行只有一份拷贝，在我自己的缓存里，所以不会有任何冲突。反之，如果有其它处理器想读取这个缓存行（马上能知道，因为一直在嗅探总线），独占或已修改的缓存行必须先回到"共享"状态。如果是已修改的缓存行，那么还要先把内容回写到内存中。**引起数据状态转换的CPU cache操作也有四种**，因为所有CPU核的数据都会经过总线，因此这里其实对应的就是CPU在读取或写入数据时对总线的数据请求类型，分别如下：

1. 本地读取（Local read），简称LR：当前CPU核读本地高速缓存中的数据
2. 本地写入（Local write），简称LW：当前CPU核将数据写到本地高速缓存中
3. 远端读取（Remote read），简称RR：其他CPU核将主内存中的数据读取到高速缓存中来
4. 远端写入（Remote write），简称RW：其他CPU核将高速缓存中的数据写回到主存里面去

状态转换和cache操作

Modify

场景：当前CPU中数据的状态是Modifiy，表示当前CPU中拥有最新数据，虽然主存中的数据和当前CPU中的数据不一致，但是以当前CPU中的数据为准；

> LR：此时如果发生local read，即当前CPU读数据，直接从cache中获取数据，拥有最新数据，因此状态不变；
> LW：直接修改本地cache数据，修改后也是当前CPU拥有最新数据，因此状态不变；
> RR：因为本地内存中有最新数据，当本地cache控制器监听到总线上有RR发生的时，必然是其他CPU发生了读主存的操作，此时为了保证一致性，当前CPU应该将数据写回主存，而随后的RR将会使得其他

> CPU和当前CPU拥有共同的数据，因此状态修改为S；
> RW：当cache控制器监听到总线发生RW，当前CPU会将数据写回主存，因为随后的RW将会导致主存的数据修改，因此状态修改成I；

Exclusive

场景：当前CPU中的数据状态是exclusive，表示当前CPU独占数据（其他CPU没有数据），并且和主存的数据一致；

> LR：直接从本地cache中直接获取数据，状态不变；
> LW：修改本地cache中的数据，状态修改成M（因为其他CPU中并没有该数据，不存在共享问题，因此不需要通知其他CPU修改cache line的状态为I）；
> RR：本地cache中有最新数据，当cache控制器监听到总线上发生RR的时候，必然是其他CPU发生了读取主存的操作，而RR操作不会导致数据修改，因此两个CPU中的数据仍和主存中的数据一致，此时cache line状态修改为S；
> RW：同RR，当cache控制器监听到总线发生RW，必然是其他CPU将最新数据写回到主存，此时为了保证缓存一致性，当前CPU的数据状态修改为I；

Shared

场景：当前CPU中的数据状态是shared，表示当前CPU和其他CPU共享数据，且数据在多个CPU之间一致、多个CPU之间的数据和主存一致；

> LR：直接从cache中读取数据，状态不变；
> LW：发生本地写，并不会将数据立即写回主存，而是在稍后的一个时间再写回主存，因此为了保证缓存一致性，当前CPU的cache line状态修改为M，并通知其他拥有该数据的CPU该数据失效，其他CPU将cache line状态修改为I；
> RR：状态不变，因为多个CPU中的数据和主存一致；
> RW：当监听到总线发生了RW，意味着其他CPU发生了写主存操作，此时本地cache中的数据既不是最新数据，和主存也不再一致，因此当前CPU的cache line状态修改为I；

Invalid

场景：当前CPU中的数据状态是invalid，表示当前CPU中是脏数据，不可用，其他CPU可能有数据、也可能没有数据；

> LR：因为当前CPU的cache line数据不可用，因此会发生读内存，此时的情形如下。如果其他CPU中无数据则状态修改为E；如果其他CPU中有数据且状态为S或E则状态修改为S；如果其他CPU中有数据且状态为M，那么其他CPU首先发生RW将M状态的数据写回主存并修改状态为S，随后当前CPU读取主存数据，也将状态修改为S；
> LW：因为当前CPU的cache line数据无效，因此发生LW会直接操作本地cache，此时的情形如下。如果其他CPU中无数据，则将本地cache line的状态修改为M；如果其他CPU中有数据且状态为S或E，则修改本地cache，通知其他CPU将数据修改为I，当前CPU中的cache line状态修改为M；如果其他CPU中有数据且状态为M，则其他CPU首先将数据写回主存，并将状态修改为I，当前CPU中的cache line转台修改为M；
> RR：监听到总线发生RR操作，表示有其他CPU读取内存，和本地cache无关，状态不变；
> RW：监听到总线发生RW操作，表示有其他CPU写主存，和本地cache无关，状态不变；

总结

Java中线程的本地内存，栈内存，局部变量这部分内存对应的硬件区域是在CPU的高速缓存或者寄存器中，而主存，堆内存，全局变量对应的是计算机的内存条中的内存或是虚拟内存。在并发情况下线程会优先逐级的从CPU的三级高速缓存中读取数据，如果高速缓存中没有数据就会从主存中读取数据。在多个线程同时需要从主存中读取统一数据的情况下，CPU总线提供了总线锁机制来保证正确的同步，当然这部分对于Java开发者来说不可能直接使用总线锁，这里JVM提供的内存模型就是对硬件机制的封装，因此只要正确使用JVM提供同步机制就可以了。语言层面就是synchronize，volatile，Lock类。

参考

1. 《Java并发编程的艺术》
2. 无锁队列Disruptor原理解析